Advanced programming

DB2 Information Management Software

`http://www-136.ibm.com/developerworks/db2`

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

What is this tutorial about?

In this tutorial, you'll learn advanced programming skills for writing applications that interface with the DB2 Universal Database. You will learn how to:

- Utilize dynamic and static SQL within programs
- Cast and use user-defined types (UDTs) within a program
- Understand compound SQL and know when to use it
- Understand concurrency considerations within an application
- Understand a distributed unit of work
- Use parameter markers
- Determine the best approaches to programming using Unicode
- Use performance enhancement features

This is the sixth in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 6 of the exam, entitled "Advanced programming." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition for reference.

Who should take this tutorial?

To take the DB2 UDB V8.1 Family Application Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 52) to prepare for that test. It is a very popular tutorial series that has helped many people understand the fundamentals of the DB2 family of products.

This tutorial is one of the tools that can help you prepare for Exam 703. You should also review Resources on page 52 at the end of this tutorial for more information.

This tutorial is designed to help you study for the advanced programming section of the DB2 Family Application Development exam. The tutorial helps you better understand how your application can interact with a database. For this tutorial to be useful, you should have good background knowledge of how a database works. You should also have a basic knowledge of:

- SQL statements
- Database objects
- Application compilation
- DB2 UDB products

Terminology review

Before you begin this tutorial, you need to be familiar with the concept of an *SQL access plan*. This is a set of steps that the DB2 engine uses to execute a SQL statement. It would include the indexes that are used, the times at which fields are retrieved from the database tables, and the order in which the steps of the query are executed. The access plan is created by a database engine based on an SQL statement sent to that engine.

The term is used throughout this tutorial, so it's important to make sure you understand it up front.

About the author

Drew Bradstock is an engagement manager for IBM Competitive Migration Services. He was previously part of the IBM Data Management Business Partner Enablement team, where he worked with IBM's partners to migrate their applications to DB2. As part of this team, he was a coauthor of the DB2 textbook *DB2 SQL Procedural Language for Linux, UNIX, and Windows.* He was also a coauthor of the IBM Redbook *Scaling DB2 UDB on Windows Server 2003*. Drew has specialized in performance-tuning databases and migrating application code and architectures.

Drew is an IBM-certified DB2 V8.1 Advanced Administrator, Business Intelligence expert and Application Developer. He was also a member of the team that developed the DB2 V8 for Linux, UNIX, and Windows (LUW) certification exams.

You can contact Drew at *drewkb@ca.ibm.com*.

Notices and trademarks

Section 2. Using dynamic and static SQL

Static vs. dynamic SQL

When you write your application, it is important that you understand how you can execute your SQL code. There are two separate methods for executing SQL, and each of these have distinct properties and behaviors. By understanding how they work, you can make the decision as to which you should use in a given application.

- **Static SQL:** This SQL is used for fixed statements, where the statement's format is known ahead of time. It is used in stored procedures and for queries that are known during application development.
- **Dynamic SQL:** This SQL is completely dynamic and can be built at run time. This allows your application to adapt to the user's input.

Comparing SQL types

There are a number of differences between the two SQL types.

	Static SQL	**Dynamic SQL**
Compile time	Compiled when the application is compiled.	Compiled when the SQL is executed.
Flexibility	SQL statements are defined at application creation.	SQL statements can be built at run time and can take any format.
Data input	Uses host variables.	Uses parameter markers.
Variable appearance	Indicated by a `:` followed by the variable name.	Indicated by a `?`.
Example	`WHERE lastname = :variableA`	`WHERE lastname = ?`

Using static SQL in programs

Static SQL's main advantage is that its total execution time is less than that for dynamic SQL. Static SQL is compiled and bound to the database when an application is built. This means that when the SQL is executed, it doesn't need to be compiled again. This can save time, especially if the SQL is quite complex and the corresponding compilation time could be long.

All of the SQL statements that are embedded in stored procedures are static SQL. These procedures are therefore compiled ahead of time, which allows the stored procedures to be executed quickly, without requiring a recompilation. Dynamic SQL is built using strings and is not embedded in the procedure code. DB2 will therefore not know the precise content of the SQL statements when the procedure is initially built. Even if you had a dynamic SQL statement based on a string with the entire SQL statement in it, the statement would still be dynamic. This is because DB2 does not read the contents of the string when compiling a procedure. Here's an example of static SQL:

```
SELECT empno,edLevel
FROM employee
```

And here's a dynamic SQL string:

```
SET sqlStr = 'SELECT empno, edLevel FROM employee'
```

Static SQL also uses the database statistics at compilation time to determine the best access plan. This can have both positive and negative impacts on your application. On the positive side, you know that the access plan is not going to change and that the SQL statement will always be executed in the same way. On the other hand, if the data in the database changes, the plans for the SQL statements will still execute based on the old statistics. The plan can later be updated by rebinding the package. This will then rebuild the plan based on any new indexes or statistics.

The following languages support static SQL:

- C/C++
- COBOL
- SQLJ (this is part of the Java language -- for more information on it, see Resources on page 52 for a link to Part 5 of this tutorial series)
- FORTRAN

Here are some common uses for static SQL:

- Stored procedures
- "Canned" queries for Management Information System (MIS) programs
- SQL statements that will be executed many times without changes

Building an application with static SQL

In this panel, we'll look at the steps required to allow your program to execute the static SQL that you wish to specify.

1. Connect to the database. Before you can do anything else, you must first have a connection to the database in your program creation script. This is required because the static SQL will actually be stored in the database.

2. Prepare the application. The code file containing the static SQL must be *prepared*. This step checks all the syntax of the SQL statements that you have in your program. An error will be returned if the syntax of any of the statements is incorrect. The SQL statements are stored as packages in the database catalog tables. These packages are then used when the statement is executed. If you use the `PRECOMPILE` command then the binding stage can be performed as part of this command.

3. Compile the application. In this step, you compile your application as you normally would.

4. Bind to the database. During this phase, the SQL packages in the application are bound into the database. If any of the tables, views, or other database objects to which the SQL statements refer are dropped or altered, then the statements will have to be recompiled. This is because the packages are bound into the database: We want to ensure that you don't receive errors at runtime on SQL statements that may no longer be correct. The bind file will contain all of the access plans for the SQL that was compiled.

Let's consider how these steps would look for an example C program.

Connection:

```
db2 connect to drewDB
```

Preparation:

```
db2 prep drewsApp.sqc bindfile
```

Compilation:

```
gcc drewsApp.c db2api.lib
```

Binding:

```
db2 bind drewsApp.bnd
```

Using dynamic SQL in programs

Dynamic SQL gives a programmer a great deal of flexibility, since no information about your SQL code is required before the program is executed. The SQL statements are created and executed at run time based on user or program input. This allows you to build SQL based on the selections a user may make in an application or other real-time data input.

A dynamic SQL statement is built by joining together one or more strings. These strings are then read by the database as a full SQL statement and executed. (We'll discuss the steps in this process in more detail in the next panel.) Every time a dynamic SQL statement is executed, it has to be recompiled. The recompilation time can be a large problem, since for short-running OLTP-type SQL statements, the compilation time can actually be longer than the run time.

One of the greatest advantages of a dynamic SQL statement is that it uses the most current statistics on the database. This ensures that the best compilation path for the SQL will be generated. Static SQL relies on old statistics, which sometimes results in poorly running SQL.

The following languages support dynamic SQL:

- C/C++
- COBOL
- The Java language
- FORTRAN
- REXX
- Visual Basic

Some common uses of dynamic SQL code include:

- Ad hoc query generation
- SQL that adapts to user input
- Installation programs
- Most Java applications (since static SQL is rarely used with the Java platform)

Building a dynamic SQL statement

In this panel, we'll look at the steps required to allow your program to execute dynamic SQL code.

1. Create the string. The actual SQL to be executed must first be written to a string. The creation of such strings can be quite complex, involving dozens of lines of code, or can be quite simple and dependent on a single `IF/ELSE` clause. The advantage to dynamic SQL is that you are only limited by your creativity. The example below shows you how to write the string in SQL PL:

```
sqlString = "Select lastname,firstname,empno from employee"
```

2. Prepare the SQL string. Once you have created the SQL string, you must prepare it. The `PREPARE` statement executes a number of steps in the database engine that ensure that the string you have given it is semantically correct. The engine then optimizes the execution of the string. `STATEMENT` objects store the information about the access plan for the SQL string. The `host-variable` in the following syntax diagram is the string that you stored the SQL in.

```
>>-PREPARE--statement-name--FROM--host-variable--------|
```

Here's an example:

```
PREPARE statementVariable FROM sqlString
```

3. Execute the SQL string. After the string has been prepared, you can execute it. Here's the syntax diagram:

```
>>-EXECUTE--statement-name-------|
```

Here's an example:

```
EXECUTE statementVariable
```

This isn't the only way to build a dynamic SQL statement, however. There is another choice you can make in the second step:

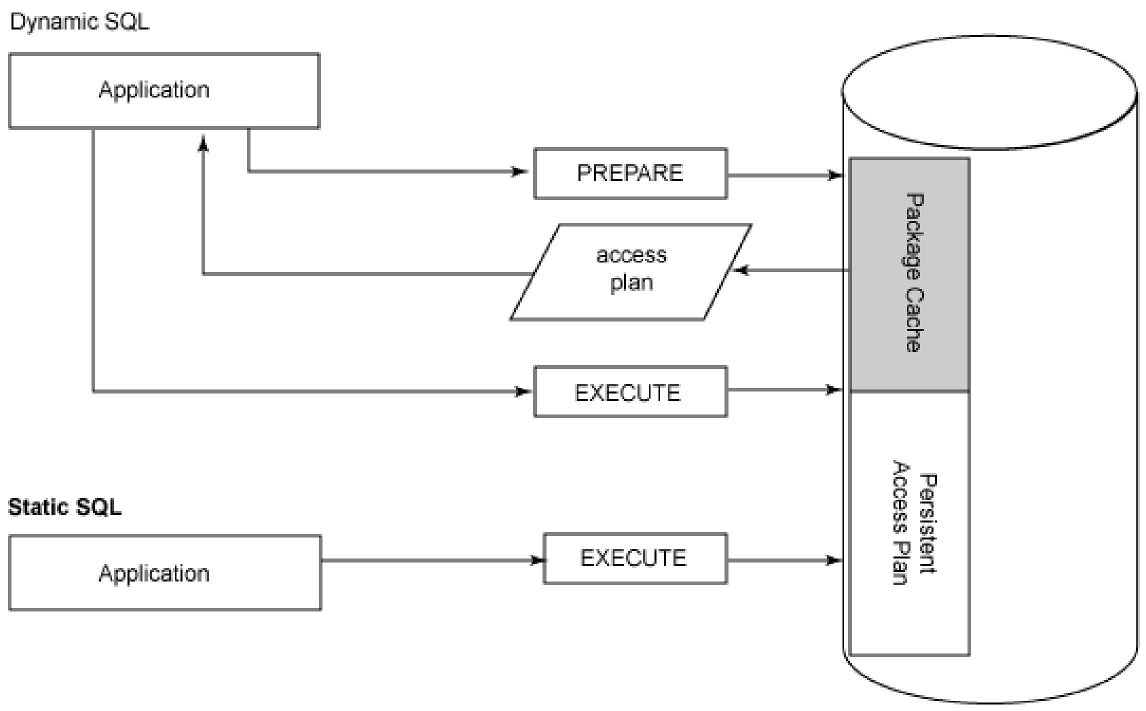
2. Execute the string immediately. You can also just execute the SQL string immediately and combine the preparation step with the execution step. This is done using the `EXECUTE IMMEDIATE` command. Here's the syntax:

```
>>-EXECUTE--IMMEDIATE--sql-statement-------|
```

Here's an example:

```
EXECUTE sqlString
```

The following figure shows the steps for dynamic and static SQL compilation.



Static and dynamic SQL: Advantages and disadvantages

There are a number of advantages to using static SQL:

- Compilation is done at program creation time.
- The SQL execution plan does not change.
- The SQL execution plan is stored in the DB2 catalog tables and can be reviewed.

There are a number of disadvantages as well:

- The SQL code cannot be changed.
- Old statistics are used to generate its access plan, so that plan could become out of date as the data changes.

There are a number of advantages to using dynamic SQL code:

- It is extremely flexible.
- It can be built based on user input.
- It uses the most recent statistics.

But dynamic SQL too has disadvantages:

- It has to be compiled at run time.

- SQL plans are not fixed and can change, which may cause unexpected performance changes.
- Building the strings can be complicated and sometimes hard to understand.

Further reference

For more information on static and dynamic SQL, search the DB2 Information Center (see Resources on page 52) for the following keywords:

- Dynamic SQL
- Static SQL
- `PREPARE` statement
- `EXECUTE` statement

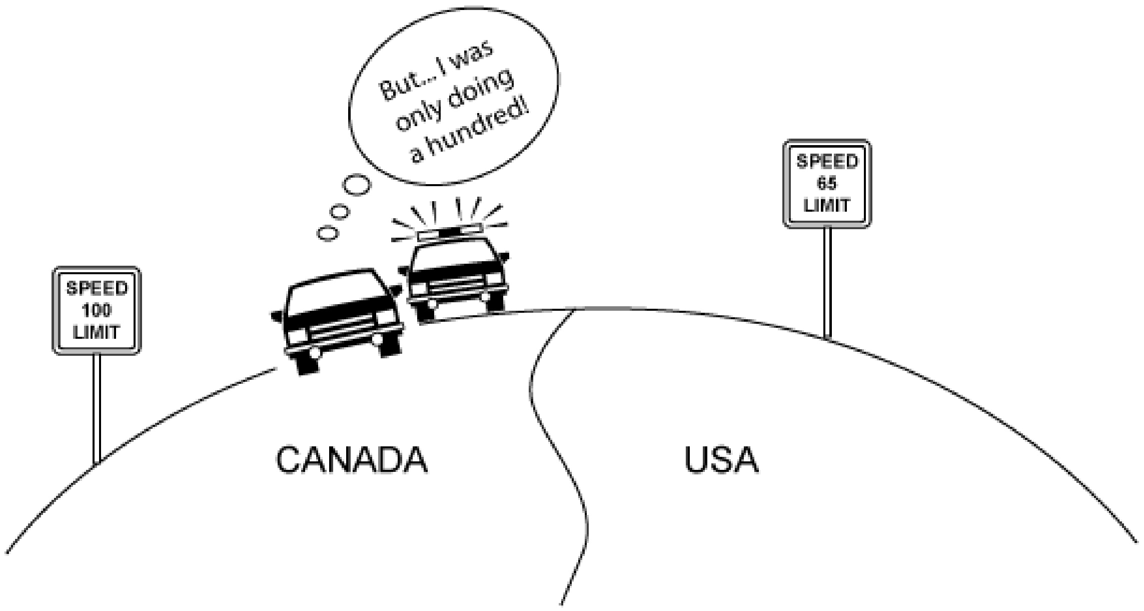
Section 3. Using user-defined types in your programs

What are user-defined types?

User-defined types, or UDTs as they are commonly known, are data types that you can create in DB2. A UDT is composed of other DB2 data types, such as `INTEGER` or `CHAR`, or even of other UDTs.

UDTs are used to establish the context of a variable you are using. It will allow you to keep track of how an object is being used in your application. You can also define the relationships between different data types and UDTs.

Let's consider an example. If you have an application that determines the best routes between stores in the US and Canada, you will most likely have to work with distances in both metric and imperial measurements. You may not know if the data in a table is stored in kilometers or in miles. You could use a UDT to create a `KM` object and another UDT for a `MILES` object. This would make it clear which type of object is stored in your table. You could also write a function that automatically adds `KM` and `MILES` objects together properly.



Creating a UDT

You can create a UDT with the statement `CREATE DISTINCT TYPE`. The syntax of the command is as follows:

```
   -CREATE DISTINCT TYPE--distinct-type-name--AS----------------->
      --| source-data-type |--WITH COMPARISONS---------------------|
```

The source data type can be any standard data type used by DB2.

Here are some examples:

```
   CREATE DISTINCT TYPE km AS INTEGER WITH COMPARISONS
   CREATE DISTINCT TYPE miles AS INTEGER WITH COMPARISONS
```

When the UDT has been successfully created, DB2 will automatically generate support for the standard comparison operators (=, <>, <, <=, >, and >=) for use with the source data type.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

° `SYSADM` or `DBADM` authority
° `IMPLICIT_SCHEMA` authority on the database (if the schema name of the distinct type does not refer to an existing schema)
° `CREATEIN` privilege on the schema (if the schema name of the distinct type refers to an existing schema)

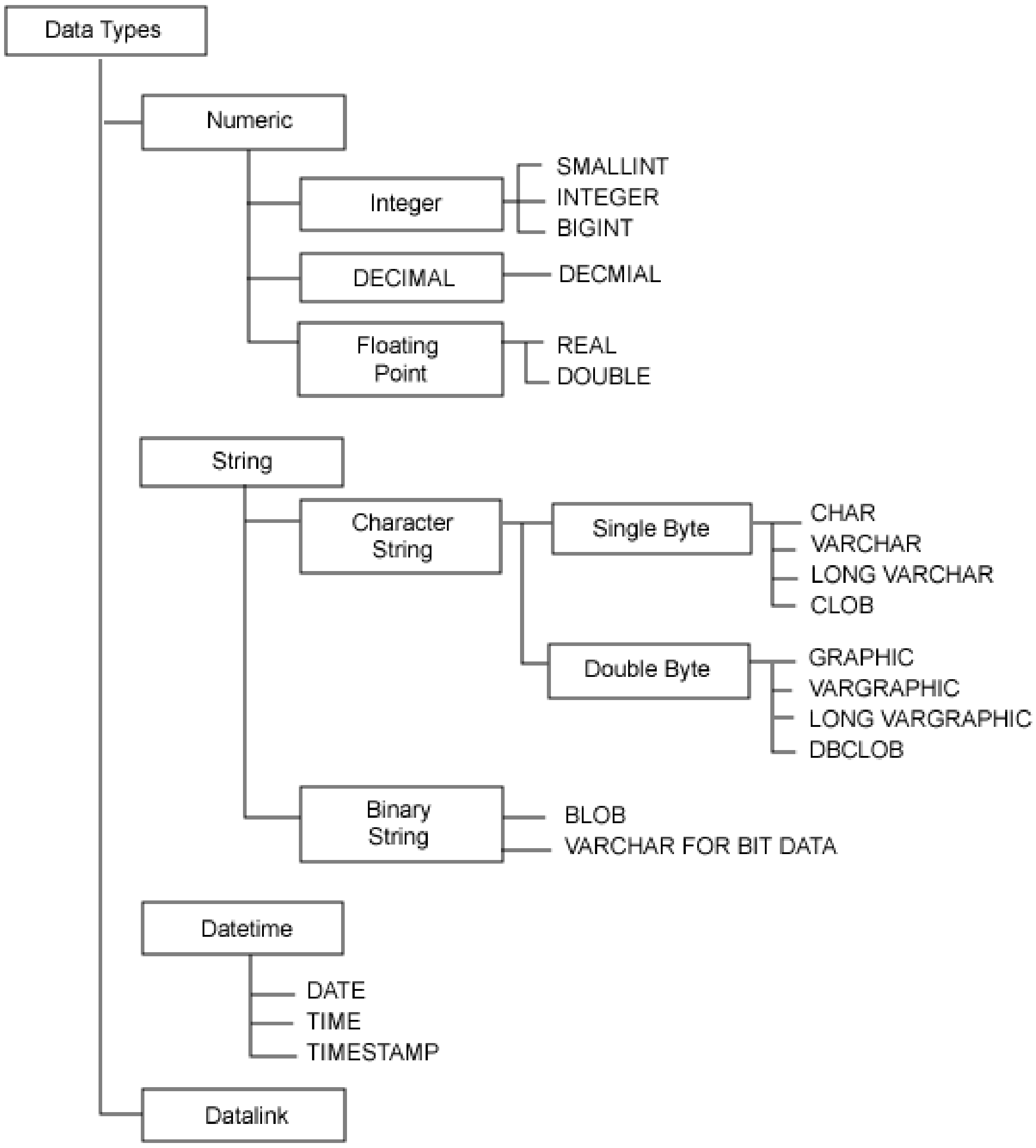
Note that UDTs cannot be used in the calling of stored procedures and functions.

Using a UDT

People often generate a lot of errors when they first begin working with UDTs. This is because UDTs interact differently with regular data types and each other. Normally, DB2 automatically translates the data type for you, but with UDTs this will not always occur.

Strong casting

DB2 will not automatically translate data types from one type to another. Some data types will be translated if they are of the same data type group, as shown in the diagram below. If they are not of the same group, then an error will occur.



One common problem that programmers encounter: Within some other RDBMSs, a character field will automatically be cast to a numeric field if mathematical equations are applied against it. If you tried to use the following code with DB2, you would receive an error.

```
empno = CHAR(6)  {but the field is always numeric}
SQL String: SELECT empno + 1 FROM employee
```

This statement would fail because you cannot add 1 to a character field, even if the field is the string representation of a number.

Casting UDTs

Even though DB2 uses strong casting, you can still perform operations on UDTs and have them interact with other data types and UDTs. If you want to compare two different UDTs, you can cast them to the same data type using the `CAST` command. Here's the syntax:

```
 CAST--(--objectName--AS--target-data-type--)----|
```

Here's an example:

```
CAST (empno AS INTEGER)
```

Let's look at a more in-depth example. First, create our distinct types and a table that uses them:

```
CREATE DISTINCT TYPE km AS INTEGER WITH COMPARISONS;
CREATE DISTINCT TYPE miles as INTEGER WITH COMPARISONS;

CREATE TABLE cityInfo (
        cityName CHAR(20),
        width_k KM NOT NULL,
        width_m MILES NOT NULL
);
```

Now, this next code snippet works because the data types are properly converted to be the same data type. The SQL statement converts the integer to a `KM` object for the comparison.

```
SELECT cityName
FROM cityInfo
WHERE width_k > km(10)
```

Now let's add some code that will cause a problem. This next snippet will not work because DB2 will not know how to compare the `MILES` and `KM` data types:

```
SELECT cityName
FROM cityInfo
WHERE width_k > width_m
```

This final snippet will work, however, because the data types have both been cast to `INTEGER`.

```
SELECT cityName
FROM cityInfo
WHERE CAST(width_k AS INTEGER) > CAST(width_m AS INTEGER)
```

Further reference

For more great information about UDTs, search the DB2 Information Center (see Resources on page 52) for the following keywords.

- `CREATE DISTINCT TYPE`
- User-defined type
- `CAST`
- Strong typing

Section 4. When to use compound SQL

Normally, SQL statements are executed individually. This allows for easy programming, but it also means that each statement and result set must be transmitted between the client and the server. To reduce the amount of network traffic that your application generates, you can instead group a number of SQL statements into a single group. This group is referred to as *compound SQL*.

There are two types of compound SQL: *atomic* and *non-atomic*.

With atomic SQL, the application receives a response from the database manager when all substatements have completed successfully or when one substatement ends in an error. If one substatement ends in an error, the entire block is considered to have ended in an error. Any changes made to the database within the block are rolled back.

With non-atomic SQL, the application receives a response from the database manager when all substatements have completed. Each substatement within a block is executed regardless of whether the preceding substatement completed successfully. The group of statements can only be rolled back if the unit of work containing the non-atomic compound SQL is rolled back.

Compound SQL: Advantages and disadvantages

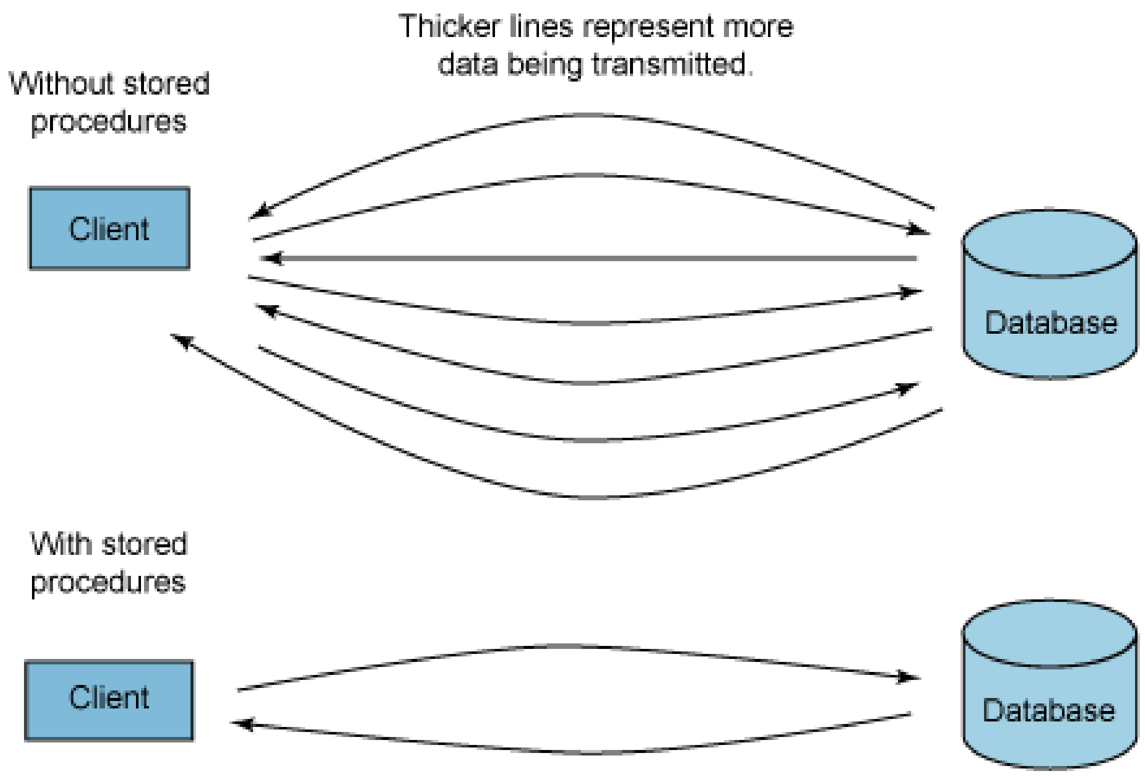
There are several advantages to using compound SQL.

- You reduce network traffic, since only the initial block of SQL and the final result set will be transmitted.
- You improve overall application speed, because more work is done on the server instead of the client.

However, there are disadvantages to using compound SQL as well.

- You can only receive one error message from the entire group of SQL statements.
- You cannot tell which SQL statement failed if an error occurs.

The diagram below is an example of the reduction in network traffic that can occur when you use use compound SQL. The thicker lines represent more network traffic. The reduction is entirely dependent on the balance of `SELECT`, `INSERT`, `UPDATE` and `DELETE` statements you have in your compound SQL.


Thicker lines represent more data being transmitted.
Without stored procedures
Client
Database
With stored procedures
Client
Database

There are several situations in which using compound SQL would be a good idea:

- You are going to be sending a great deal of data back and forth as part of a block of SQL statements. Using compound SQL will reduce network traffic.
- You have a slow client and want to utilize the high speed of the server. Using compound SQL in a stored procedure ensures that the processing will occur on the server and that only the final result is transmitted back.

Using compound SQL

Compound SQL is used in stored procedures and also with the following languages:

- Embedded SQL
- DB2 Call Level Interface (CLI)
- JDBC

Dynamic compound SQL

Dynamic compound statements are compiled by the database engine as a single statement. This statement can then be used for sections of the code that require little control flow logic but may have a great deal of data flow between steps. By making the statement compound, you would reduce the data needed to be transmitted back and forth for each step.

For larger and more complex statements, however, it is a better strategy to use stored procedures.

A compound statement block begins with the `BEGIN` keyword and ends with the `END` keyword. Here's an example:

```
BEGIN ATOMIC
    UPDATE employee
        SET salary = salary * 1.1
        WHERE edlevel > 12;

        UPDATE dept
        SET budget = budget * 0.8
        WHERE profitMargin > 10;

        INSERT INTO deptList (empno,salary)
                VALUES (SELECT empno,salary
                                FROM employee WHERE edlevel = 15);
END
```

If the statement listed above failed at the first `UPDATE` statement, you would not be able to tell which of the statements failed. The advantage to making this a compound statement is that the only data being transmitted to the server and back are the SQL statements and the final SQL information.

Further reference

You can search the DB2 Information Center (see Resources on page 52) for the following keywords for more information about compound SQL.

- Compound SQL guidelines
- Stored procedures
- `BEGIN ATOMIC`

Section 5. Working with DB2 database concurrency

DB2 concurrency

Every database programmer should understand how database concurrency works. RDBMSs all follow core database data integrity principals, but they vary a great deal in how they handle locking, transaction, and concurrency control. By learning more about DB2 concurrency, you can both avoid errors and concurrency problems and also potentially improve your application performance greatly with only small changes.

To avoid confusion later on, it's important that we define a few terms first:

° **Isolation level**: DB2 uses different isolation levels to maintain and control the integrity of the data in the database. The isolation level defines the degree to which a process is protected (or *isolated*) from changes made by other applications as they also execute in the database.

° **Lock mode**: This represents the type of access allowed for the lock owner as well as the type of access permitted for concurrent users of the locked object. This is sometimes referred to as the *state* of the lock.

° **Deadlocks**: This is a condition that occurs when two or more separate processes are waiting for resources that the other process(es) in the deadlock have locked. Neither process can progress because each process is waiting on the others.

° **Lock escalation**: This occurs when a lock is escalated from a smaller lock granularity to a higher one due to insufficient lock memory space. This could, for instance, mean escalating a row lock to a table lock.

Isolation levels: UR and CS

There are four main isolation levels used by DB2. The isolation levels are listed on this panel and the next in order from the level offering the least data protection to the one offering the most.

1. Uncommitted read (UR). Uncommitted read is the weakest of the isolation levels and offers the least data protection. Uncommitted read will read any row during a unit of work, even if it is being changed by another application process. These changes do not even need to have been committed. This means that your application may be returning data that would no longer be valid if another application's transaction were rolled back. UC implies the possibility of reading incorrect data and ignoring all locks, which is why it is sometimes referred to as *dirty read.*

Uses: UC is quite useful in large data warehouse systems where you may not want to wait on a few locks that could be held in a generally read-only environment. It is commonly used for reporting purposes to avoid all locks in the database.

2. Cursor stability (CS). Cursor stability is the default and most commonly used isolation level. CS holds a lock on any row on which the cursor is currently positioned. This ensures that the row cannot be changed by another process as it is being read or altered. The lock on the row is held until another new row is fetched or until the unit of work is terminated or completed.

If any row is updated, the lock will be held until the unit of work is completed. The CS isolation level will not return uncommitted data.

Uses: The CS isolation level is the most commonly used because it ensures that you do not read any rows that are currently locked and it also locks the row that you are reading. Because the lock only holds the row currently being processed, contention is minimal and locking is reduced, but data protection is still enforced. Most common transaction processing systems use this isolation level.

Isolation levels: RS and RR

3. Read stability (RS). Read stability will lock all rows that an application retrieves during a *unit of work* (UOW). This ensures that any of the data that it works with cannot be altered and that the result set will remain constant during the UOW. The RS isolation level will not be able to read any uncommitted data. Read stability ensures that other applications cannot change data that you are returning or working with during the unit of work.

Uses: This isolation level offers greater data protection, since all rows that are returned by a query are locked during the UOW. However, it does greatly increase the number of locks being held and can negatively affect concurrency.

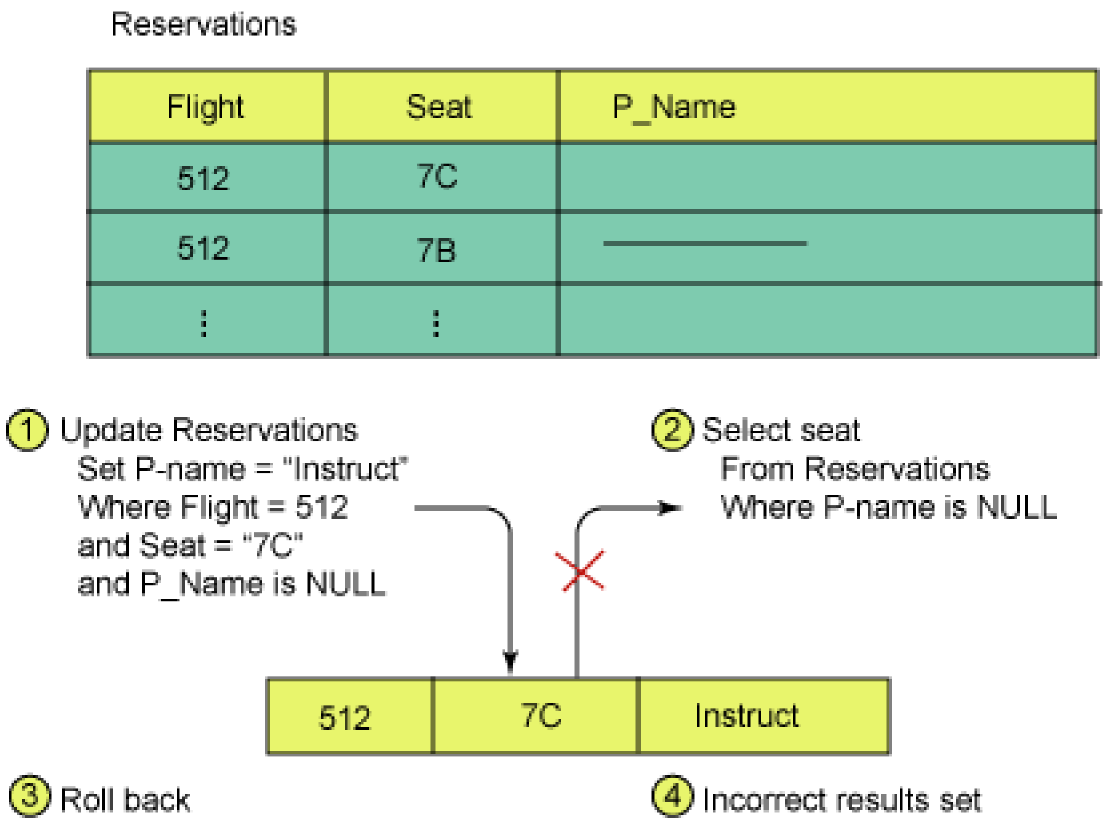
4. Repeatable read (RR). This is the strongest isolation level; it offers the greatest degree of data integrity. Locks will be held on all rows that are processed during the running of a query. This means that rows that are not even returned in a result set will be locked until the unit of work is completed. This has an extremely negative impact on concurrency, since often entire tables are locked by this isolation level. It does however ensure that if a query is repeated during a unit of work, the exact same result will be returned. That is where the term *repeatable read* comes from. The largest penalty of this isolation level is that table locks are often taken by the optimizer. This effectively stops any other application from processing queries on the table that is locked. No other application would be able to update, delete, or insert a row into the table until the RR unit of work is done.

Uses: This isolation level is used when queries must return the exact result every single time throughout a query, and no possibility of changes to the data set are acceptable.

Common data integrity issues

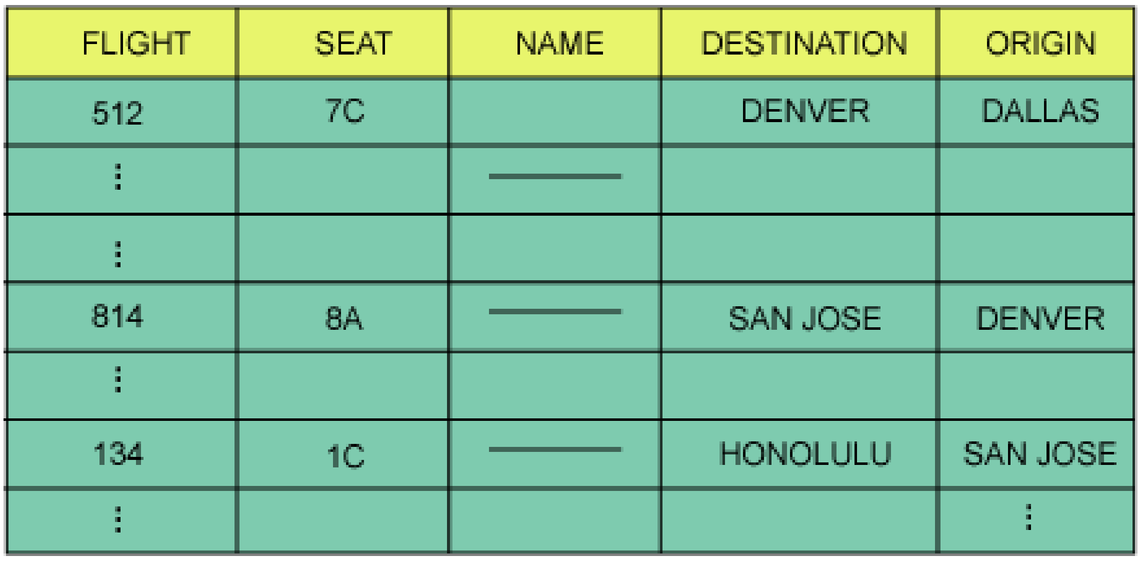
DB2's system of isolation levels is designed to avoid data integrity problems by preventing data from being accessed by other applications. On this panel, we'll look at some potential data integrity problems.

Uncommitted data. This occurs when data can be read that has been altered or inserted but not committed.



In the example above, a row is updated and the P-name field is changed. A SELECT query is executed by another application using Uncommitted Read. Since UR ignores the locks on the row that was changed, it will return the altered value. However, the update transaction is rolled back before it is committed. The SELECT query has therefore returned an incorrect row.

Nonrepeatable reads. These occur when SELECTs return different result sets within the same result set.

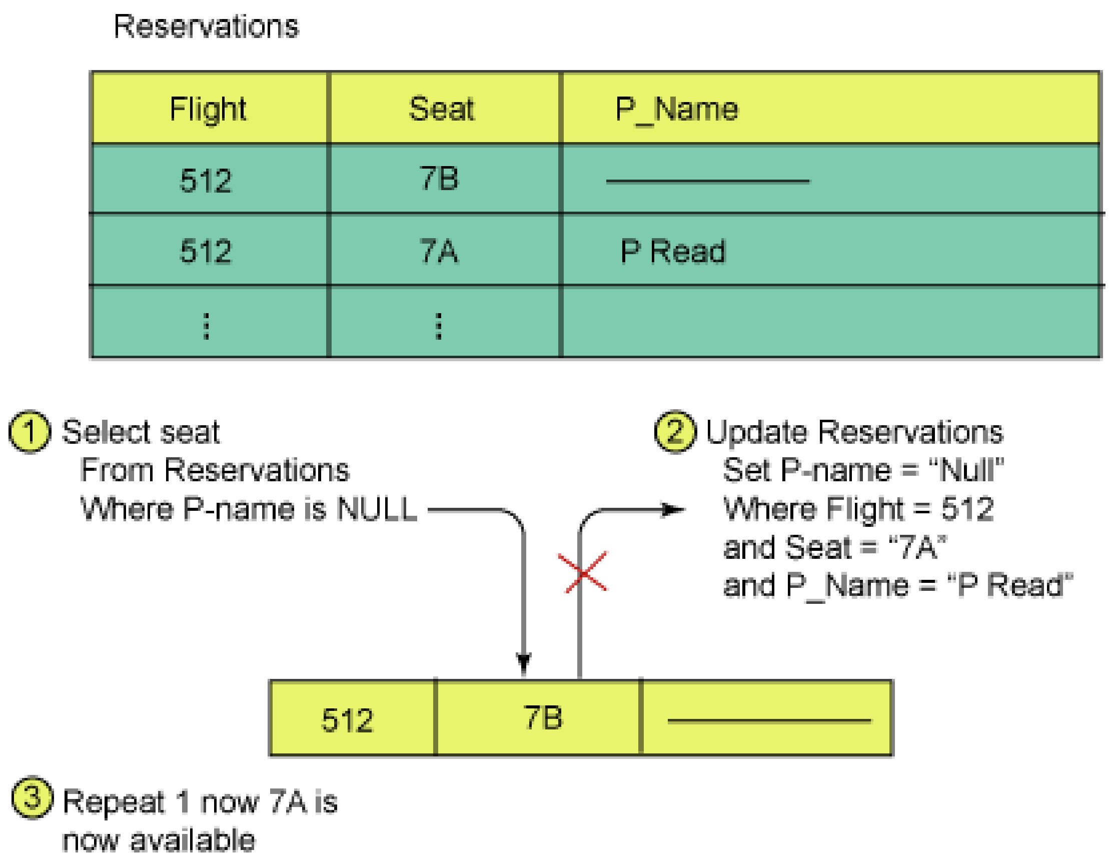


FLIGHT	SEAT	NAME	DESTINATION	ORIGIN
512	7C		DENVER	DALLAS
⋮		—		
⋮				
814	8A	—	SAN JOSE	DENVER
⋮				
134	1C	—	HONOLULU	SAN JOSE
⋮				⋮

Book a flight from Dallas to Honolulu

In the example above, an application A looks up the route from Dallas to Honolulu. The transaction is repeatable, so no matter when in the transaction it is executed, it will return the same result. However, if an application B added a row to the flight table with a direct flight from Dallas to Honolulu, then a different result could occur in application A depending when the query is executed. This means that if application B is allowed to alter the table while application A is running its transaction the query is therefore nonrepeatable.

Phantom reads. These occur when an application executes the same SQL code twice in a transaction and the second execution returns additional rows.



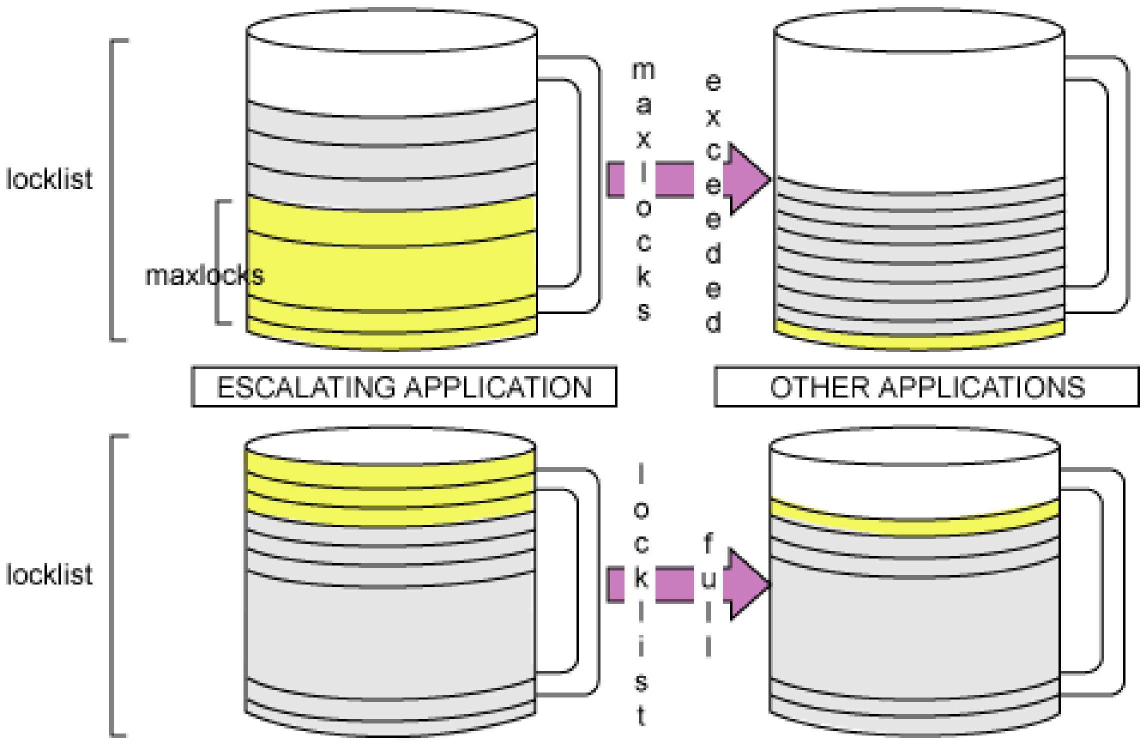
Data integrity protection

Not all applications will suffer from the problems listed on the previous panel. If you think that these scenarios could occur in your application, you should know which isolation levels protect against them. The following table can help you decide.

Isolation level	**Access to uncommitted data**	**Nonrepeatable reads**	**Phantom reads**
Repeatable read	Not possible	Not possible	Not possible
Read stability	Not possible	Not possible	Possible
Cursor stability	Not possible	Possible	Possible
Uncommitted read	Possible	Possible	Possible

Lock escalation

DB2 keeps track of all the locks that are held by the database in a memory area called the `LOCKLIST`. Each lock in DB2 takes up a very small amount of memory (36 to 112 bytes), so it is rare that a database runs out of `LOCKLIST` memory. However, if this occurs, some of the locks will be escalated. This involves increasing the size of a lock to the next level of locking. The escalation would be from either a row lock to a table lock. Increasing the lock size can cause a lot of concurrency problems, since millions of rows in a table -- instead of just a few -- could suddenly be locked by an application.



The first image illustrates an example where an application has exceeded the percentage of the total lock memory that it is allowed to have (the `MAXLOCKS` value). The application therefore has its locks escalated and the total amount of space allocated to it reduced. The second image illustrates the more common occurrence of the total lock memory limit being reached. The lock escalation problem can be overcome by increasing the `LOCKLIST` size. However, if you continue to run into locking problems, you should probably look to see if you are failing to commit your transactions and therefore not freeing up locked rows.

Lock types

There are 12 types of locks currently used by DB2 V8.1, but it is more important to understand how locking works than to memorize all the lock types. The *lock mode* is the type of access allowed for the lock owner as well as the type of access permitted for concurrent applications on the locked object.

The two major lock types are *exclusive* and *share* locks. Share locks, or S locks, are held on rows that are being processed for reading; they prevent the row from being deleted or altered as the row is being read by the database engine. Once the row is no longer being read, the lock is released and can be changed. Exclusive locks, or X locks, are held on rows that have been updated, deleted, or inserted. The exclusive lock prevents the row from being read or updated until the transaction has been committed or rolled back. This is done so uncommitted data is not read by another application.

All lock types are ignored by the application that holds them, since the locks are only used to prevent *other* applications from changing or accessing data that has been altered or changed by the application holding the locks. The different lock modes all interact with each other differently, as you'll see in the next panel.

Lock mode compatibility

The different lock modes all interact with each other differently. A complete listing of the lock types can be found in the DB2 Information Center (see Resources on page 52). This list includes the precise definition of all the lock types and a description of when exactly each would occur.

The following list will get you acquainted with the most important types of locks.

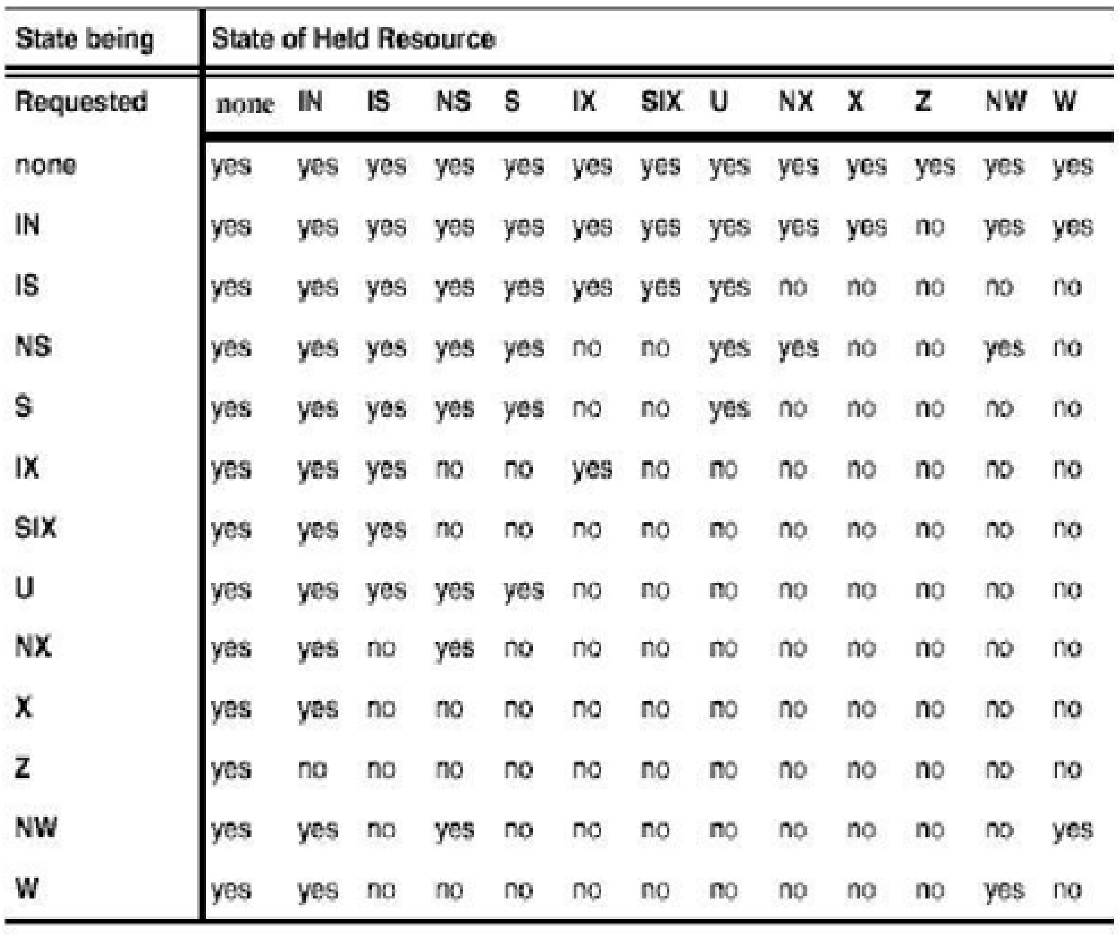
Row locks

- Share: Occur during reading of data using `SELECT`s
- Update: Occur when a row is updated using a cursor with the `FOR UPDATE` clause
- Exclusive: Occur on a row when it is inserted, updated, or deleted

Table and tablespace locks

- Intent none: The holding application can only read data, including uncommitted data; other applications can both read and update the object
- Intent share: Holding applications can only read the data, other applications can both read and update the object.
- Intent exclusive: The lock owner can both read and update the data, as can other applications.
- Update: The application can update. Other applications can read the data, but cannot attempt to update it.
- Exclusive: The lock owner can both read and change the object. Other applications can only read the data if they are using the Uncommitted Read isolation level.

The following table outlines the compatibility among different types of locks (see the DB2 Information Center for more on the abbreviations):



State being	State of Held Resource												
Requested	none	IN	IS	NS	S	IX	SIX	U	NX	X	Z	NW	W
none	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
IN	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
IS	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no	no	no
NS	yes	yes	yes	yes	yes	no	no	yes	yes	no	no	yes	no
S	yes	yes	yes	yes	yes	no	no	yes	no	no	no	no	no
IX	yes	yes	yes	no	no	yes	no	no	no	no	no	no	no
SIX	yes	yes	yes	no	no	no	no	no	no	no	no	no	no
U	yes	yes	yes	yes	yes	no	no	no	no	no	no	no	no
NX	yes	yes	no	yes	no	no	no	no	no	no	no	no	no
X	yes	yes	no	no	no	no	no	no	no	no	no	no	no
Z	yes	no	no	no	no	no	no	no	no	no	no	no	no
NW	yes	yes	no	yes	no	no	no	no	no	no	no	no	yes
W	yes	yes	no	no	no	no	no	no	no	no	no	yes	no

Deadlocks

A *deadlock* is a very serious concurrency issue, and its cause can often be difficult to determine. A deadlock occurs when one application places locks on some objects in the database and then requests to put locks on other objects. These objects are already locked by another application which in turn wants locks on objects already locked by the original application. The two (or more) applications will wait forever for each other's locks to become free .

A deadlock can end when the lock timeout value is reached and the application waiting on the locks the longest gives up and rolls back its transaction. The deadlock can also be released if it is detected by the DB2 *deadlock daemon*, which is an asynchronous process that runs at regular intervals and checks if there are any deadlocks. The daemon will choose which process to roll back.

Reducing deadlocks

Deadlocks can be corrected by setting the lock timeout variable, `LOCKTIMEOUT`, to a shorter interval. This ensures that applications do not wait as long in a deadlocked state, but will still not fix your deadlocks. The interval at which the deadlock check daemon is activated, `DLCHKTIME`, can also be decreased to increase the frequency at which deadlocks are checked. This is not recommended, since the daemon can be quite CPU intensive if there are a large number of locks being held.

Reducing lock wait and avoiding deadlocks

Often, deadlocks or lock wait can be completely avoided by small changes to the way your application handles data and transactions.

Committing transactions

One of the biggest causes of lock contention is a failure to commit transactions. If a transaction is not committed or rolled back, its locks will remain on all the objects it handles. Over time, this can cause a lot of contention with other applications. A `COMMIT` statement should be issued as soon as a business unit of work is completed. If one group of SQL statements is not dependent on the results of an earlier group, and the failure of one will not affect the results of another, then they are most likely two separate transactions. By placing a `COMMIT` between them, you free the locks from the first set of SQL statements. This reduces the likelihood of other applications having to wait to access them.

Data contention

Lock contention can also occur if one application wants to have a lock on the resources that another application has already locked. Often lock contention, and therefore lock wait, will be unavoidable. Sometimes, though, this lock contention is due to DB2 reading in too many rows, which is a result of an index not existing on the table. The lack of an index causes a table scan, and DB2 has to wait on rows that have been locked by another application, even if the locks are not involved in the transaction. This is sometimes hard to detect if the application has been designed to avoid data contention, but the index that the application expects is missing.

Most lock problems occur because exclusive locks have been left uncommitted. If you are having locking problems, it is best to look for long-running transactions that have changed data but have not been committed. You can then check to see if those transactions can be split into more than one transaction and have some of the locks released by using a commit.

Lock avoidance

In DB2 V8.1 Fixpak 4, a new locking strategy was added. To improve concurrency, in some situations DB2 now permits the deferral of row locks for CS or RS isolation scans until a record is known to satisfy the predicates of a query. By default, when row locking is performed during a table or index scan, DB2 locks each row that is scanned before determining whether the row qualifies for the query. To improve the concurrency of scans, it may be possible to defer row locking until after DB2 determines that a row qualifies for a query.

To take advantage of this feature, enable the `DB2_EVALUNCOMMITTED` registry variable. At the server command line, you would have to issue the following command:

```
db2set db2_evaluncommitted=YES
```

By turning on the new lock setting, you will no longer have to wait when scanning an index or table for rows that are locked by other processes but are not part of the final result set. This will help improve concurrency, since this has traditionally been a concurrency problem when migrating an application to DB2 from other RDBMSs.

A complete description of the new locking mechanism, along with a series of in-depth examples, can be found in the Fixpak 4 documentation or in the documentation for any Fixpak after 4.

Further reference

You can search the DB2 Information Center (see Resources on page 52) for the following keywords for more information about concurrency.

- Isolation levels
- Lock types
- Concurrency control
- Locking type compatibility
- Lock attributes

Section 6. Distributed units of work

What is a distributed unit of work?

A *remote unit of work* (RUOW) allows a user or program to update or read data at a remote location for a unit of work. Only one database can be accessed within the unit of work. The application could update multiple units of work; but only one database can be accessed during one unit of work. The characteristics of a RUOW are:

- Multiple requests (SQL statements) per unit of work are supported.
- Multiple cursors per unit of work are supported.
- Each unit of work can access only one database.
- The application program either commits or rolls back the unit of work. In certain error circumstances, the database server or DB2 Connect may roll back the unit of work.

A *distributed unit of work* (DUOW), also known as a *multisite update,* uses more than one database server within a unit of work. The characteristics of a DUOW are:

- More than one database management server is updated per unit of work.
- The application directs the distribution of work, and initiates commit.
- There may be multiple requests per unit of work.
- There is one database management server per request.
- Commitment is coordinated across multiple database servers.

How is a DUOW used?

Distributed units of work (DUOWs) give you a number of options in your queries and your applications. This is because with a DUOW, you can query across multiple databases instead of being restricted to running queries on the objects in only one. In fact, you can access more than just DB2 databases. Using the *federated* capabilities of DB2, you could have a query access DB2, Informix, and Oracle tables all at the same time.

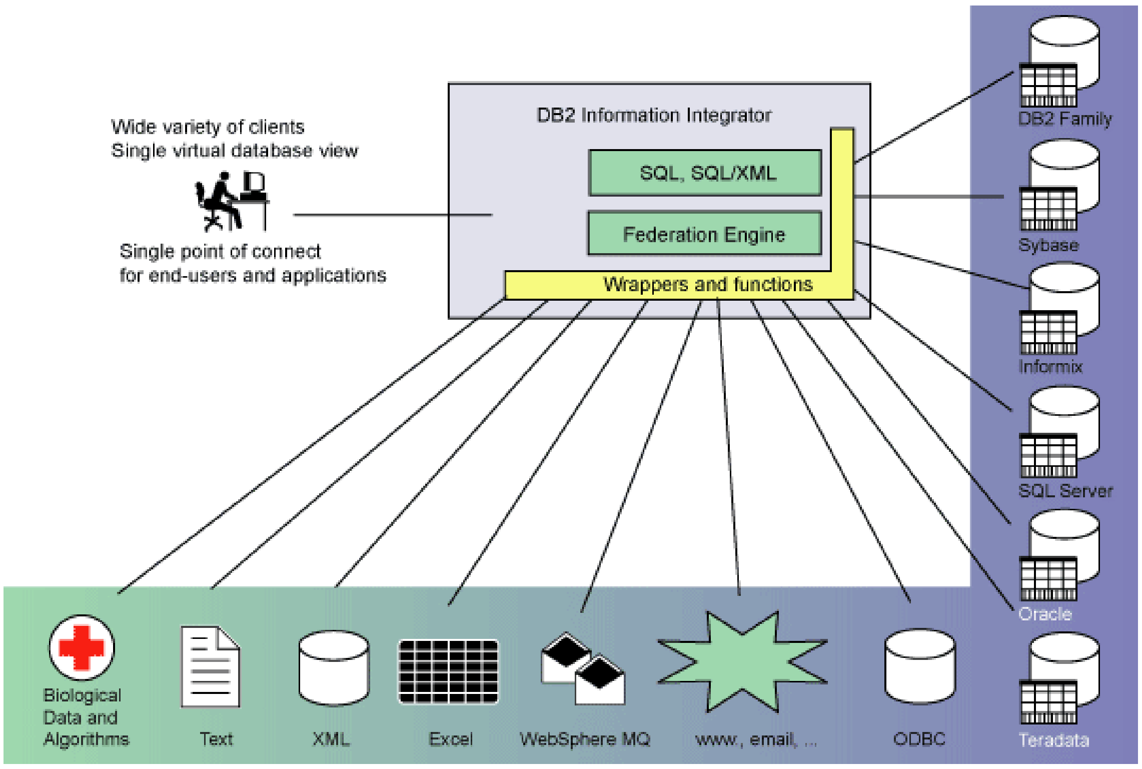
A *federated database* contains references to one or more remote data sources and their characteristics. A *nickname* is a reference to a federated data source. This nickname is referred to whenever you need to access the data source.

Let's look at an example in which we'll create and use a nickname. We'll create a nickname, `farawayTable`, for a data source called `ORA`, which contains the object `DREW.DATATABLE1`. Once the nickname is created, we can treat it like a regular database table.

```
CREATE NICKNAME farAwayTable FOR ora.drew.datatable1

SELECT * FROM ora.drew.datatable1
{this will fail since you have to refer to the table by its nickname}

SELECT * FROM farAwayTable
{this will return the results from the table}

SELECT *
FROM farawaytable,
        dept
WHERE dept.id = farawayTable.id
{this will join the two tables together and return the result set}

INSERT INTO farawayTable
VALUES (SELECT * from dept)
{this will insert all the data from the dept table into the farAwayTable}
```

The query does not have to know whether the object it refers to is part of the local database or another RDBMS entirely. If the underlying object is moved, the application does not have to be changed. This is known as *location transparency.*

The following image illustrates the potential sources and targets for federation:



Federation can also be used to reference nonrelational data sources, such as text or XML files. To do this, you have to create a *wrapper* that tells DB2 how to `SELECT`, `UPDATE`, `DELETE`, and `INSERT` from the data source. Once these are created, you can treat the data source as if it were a table in DB2. The scientific community has found this technique quite useful for working with DNA information in flat files.

Working with DUOWs

There are a number of advantages to working with DUOWs:

- Multiple data sources can be accessed easily.
- Application complexity is greatly reduced.
- Application read/write capabilities are given to more than one RDBMS.

There are a number of disadvantages as well:

- More than one database needs to be maintained.
- Error handling is more complex due to interaction with multiple systems.
- Remote systems could hang, causing SQL to take a long time to respond.
- Network transfer time of large amounts of data can greatly slow queries.
- Concurrency issues of locking of remote and local systems due to delays could cause problems.

DUOWs and federation are commonly used in the finance and pharmaceutical fields to bring together multiple data sources with one simple program. You can write a single SQL statement that returns and aggregates data from multiple sources. Also, if the underlying data source changes, only the nickname has to be altered, not the entire data access application.

Here are some specific areas where DUOWs come in handy:

- Combining reports from multiple databases
- Combining data from multiple RDBMSs
- Moving data from one RDBMS to another
- Working with nonrelational data sources in SQL statements

Further reference

Search the DB2 Information Center (see Resources on page 52) for the following keywords for more information about DUOWs.

- Multisite update
- Distributed unit of work
- Nickname
- Federated
- Information integrator

Section 7. Working with parameter markers

What are parameter markers?

Parameter markers, also called *host variables*, are used to replace the variable values in SQL statements. This is done to reduce the complexity of your applications and also to reduce the number of times that a statement has to be compiled.

Consider this example:

```
SELECT lastname, empno
FROM employee
WHERE edlevel = 14
```

If we hardcoded the value of the education level (`edlevel`) that we wanted to return information about, we would have to write a new SQL statement for each value we searched on. If we ran the query again but searched for level 13, DB2 would have to recompile the statement and store another copy of it in the catalog tables (if it was static SQL code).

Picture how much space and wasted time this would take if we were using a field like `lastname` instead and there were thousands of values. To avoid this problem, you can use a place marker in your query instead of the value. Here's how you would do it for static SQL:

```
SELECT lastname, empno
FROM employee
WHERE edlevel = :varEd
```

And here's how you would do it for dynamic SQL:

```
SELECT lastname, empno
FROM employee
WHERE edlevel = ?
```

Now that you have used a parameter marker, the SQL statement will only have to be compiled once instead of once for each value. You can also use multiple parameter markers in your queries. You can imagine how often you would have to compile the statement if you had to do it once for every combination of parameters! Here's an example:

```
SELECT lastname, empno
FROM employee
WHERE edlevel = ?
    AND salary > ?
```

How to use parameter markers

Parameter markers are used as frequently in static SQL statements as in dynamic SQL statements. Using parameter markers in static SQL reduces the number of SQL statements that need to be compiled and stored in the catalog tables. You may be wondering why you would use parameter markers in your application if you are using dynamic SQL and already have to build the string each time. We'll cover this in more detail in the next panel.

Keep in mind that the only part of an SQL statement that a parameter marker can replace is a variable value, as in the following code:

```
SELECT firstname, lastname
FROM employee
WHERE empno = ?

SELECT firstname, lastname
FROM employee
WHERE hiredate > ?
    AND mgrno = ?
```

Let's look at three examples that demonstrate common mistakes developers make when using parameter markers. Here's the first:

```
SELECT ?, lastname
FROM employee
WHERE empno = ?
```

This first example specifies a field in the `SELECT` list with a parameter marker. DB2 needs to know what fields are going to be returned in order to optimize a query. The fields that are returned have a big impact on how DB2 retrieves the data. For instance, they can determine things like the index that DB2 will use, or whether a table scan will be used instead.

Let's take a look at another example of bad code:

```
SELECT firstname, lastname
FROM ?
WHERE empno = ?
```

This second example specifies the name of the table in the `FROM` clause with a parameter marker. DB2 would be unable to determine the table from which the data is being extracted, so there is no way the SQL statement could be compiled.

And now, our third example:

```
SELECT firstname, lastname
FROM employee
```

```
WHERE ? = 14
    AND ? = ?
```

This example specifies the field that the value 14 will be checked against. It also uses two parameter markers to specify both the predicate field and the value field. DB2 would be unable to determine which was the predicate or the value, so it could not compile the statement.

Parameter markers: Advantages of use

Using parameter markers can offer a number of advantages, chief among them is reduced compilation time. DB2 first prepares a SQL statement before it can be executed. The preparation phase creates the access plan that the SQL will use to execute the SQL. When the SQL is being compiled, DB2 checks a section of DB2 memory called the *package cache.* The package cache contains all of the SQL statements and their associated access plans. If an SQL statement matches the query that you are trying to compile, then DB2 will not have to recompile it. It will instead use the access plan stored in the package cache.

Keep in mind that the SQL statement will have to match exactly for the access plan in the package cache to be reused. Let's look at some examples to illustrate what would match and what would not. Imagine that this is the original statement in the package cache:

```
SELECT deptName, location
FROM department
WHERE mgrno = '000056'
```

This would be a matching statement:

```
SELECT deptName, location
FROM department
WHERE mgrno = '000056'
```

None of these would be considered matching statements:

```
SELECT deptName, location
FROM department
WHERE mgrno = '000055'

SELECT deptName, location
FROM department
WHERE mgrno = ?

SELECT deptName, location
FROM department
WHERE mgrno = :varMgrno
```

The first unmatched statement fails because the `mgrno`, `'000055'`, is not identical to `'000056'`. DB2 uses a straight text match to see if the fields are identical. The second unmatched statement fails because a parameter marker is used instead. Using a text match, the ? symbol does not match the value `'000056'`. This third example fails for a similar reason, but a host variable was used instead of a parameter marker.

If, however, a parameter marker had been used in the original statement, then any execution of the statement using a parameter marker would match. The SQL statement using the parameter marker could then be executed using any value for `mgrno`, such as `'000055'`, `'005600'`, `'000123'`, and so on, and the statement would still only need to be compiled once.

To illustrate more completely, imagine that this is the original statement in the package cache:

```
SELECT deptName, location
FROM department
WHERE mgrno = ?
```

The following statement would match it:

```
SELECT deptName, location
FROM department
WHERE mgrno = ?
```

The advantage of not having to compile becomes clear when you consider how long a query may take to compile. Imagine that we have a query with a compile time of 0.001 seconds and an execution time of 0.001 seconds. How long will it take to execute that query 10,000 times? Here's the formula to determine that:

```
Total time = compile time + execution time
```

Using parameter markers, we have this result:

```
    0.001 + (10000 * 0.001) = 10.001 seconds
```

If we didn't use these parameter markers, we'd get this result:

```
        (10000 * 0.001) + (10000 * 0.001) = 20 seconds
```

Thus, in this case, using a parameter marker cuts our total query time in half!

Parameter markers: Disadvantages of use

Using parameter markers or host variables can have downsides too. The chief disadvantage is a lack of information about parameter values. When you use a parameter marker, DB2 does not know what the value of the parameter is when it compiles the statement. At execution time, it will substitute the specified variable value in for the parameter marker. This can cause problems if the value of the variable has a big impact on the SQL access plan chosen.

As an example, consider a table named bigTable that contains information about toys. There are 50,000 red toys, 5 brown toys, 800,000 pink toys and 75,000 green toys. The application makes queries about how many toys are being sold. Here's the table description:

```
CREATE TABLE toyLists (
        id INTEGER NOT NULL,
        colour CHAR(10),
        type CHAR(10),
        price INTEGER,
        amount INTEGER
);
```

Our table has an index on `colour`.

These two queries illustrate the way in which using parameter markers could make a big difference when working with this table:

```
SELECT price, amount
FROM toyLists
WHERE colour = 'PINK'
        AND type = 'CAR';
*********
SELECT price, amount
FROM toyLists
WHERE colour = 'BROWN'
        AND type = 'CAR';
```

In the first query, 800,000 records are going to be returned for the colour `'PINK'`. This list will then be reduced further based on how many of these pink toys are of type `'CAR'`. It would make sense to just scan the table instead of using an index, since so many records are returned.

The second example returns only 5 records based on the colour `'BROWN'`, and possibly less than that as it narrows its search to the type `'CAR'`. It would be best to use an index on `colour` to reduce the number of items you have to search for the type `'CAR'`.

Now, let's see what happens if we try to do a similar search using a parameter marker:

```
SELECT price, amount
FROM toyLists
WHERE colour = ?
        AND type = 'CAR';
```

Here, DB2 will have no idea what colour is being referred to. It doesn't know if 5 or 300,000 records are going to be returned. The different color values could have created entirely different access plans. Since a parameter marker was used, only one access plan will be created.

When parameter markers are involved, DB2 uses the *average statistics* for the variable to generate an access plan. DB2 determines how many records on average each colour has. Since `'PINK'` is so common, the average number of records per colour would be quite high, and thus DB2 would probably use a table scan for every query on colour. This would be great for `'PINK'`, but bad for `'RED'` or `'BROWN'`. Thus, it would probably have been better to not have used a parameter marker in the query, since the extra compilation time could be a lot less than the extra processing time that results from the improper use of a table scan.

When to use parameter markers

There are a number of situations in which parameter markers can improve your application. You should include parameter markers in:

- Queries that are executed multiple times and only have their parameter value change
- Queries for which the compilation time can be longer than the execution time
- Short, OLTP-type queries

However, there are some queries that do not benefit from the use of parameter markers:

- Large, OLAP-type queries where compilation time is a fraction of the total run time
- Queries where the values of the variables have a big effect on the way the SQL access plan would be created
- Queries where the parameter variable's values may be related and result in DB2 being able to recognize a pattern or data distribution from the statistics, such as (state, city) or (year, month) pairs
- SQL that runs against data sets with nonuniform distributions since using parameter markers may cause the DB2 optimizer to miss out on important statistical skews for certain values

Further reference

Search the DB2 Information Center (see Resources on page 52) for the following keywords for more information about parameter markers.

- Parameter markers
- Host variables
- Package cache
- `PREPARE`
- SQL compilation

Section 8. Moving to Unicode

What is Unicode?

When databases were first invented, everything was stored in them in standard English character sets. This could have been ASCII for UNIX or Windows platforms or EBCDIC for mainframe computers. This was fine when the data was not stored in multiple languages. As the demand grew for other languages, specific coding sets were created for non-English languages. You could then create a database specifically for Japanese- or Russian-language data.

In today's on-demand world, a company seldom has the luxury of storing data in only one language. Many databases store information from all over the world, and this demands support for all languages. The Unicode language standard was created to allow all languages to be stored as one encoding set.

Unicode is an encoding that translates a series of bytes to a character in the native language. Unicode is much more complex than ASCII or EBCDIC, because more than one byte can be combined together to create a single character that would appear on screen. The standard is very complex and has been worked on by programmers and linguists from across the globe. More information can be found at the Unicode home page (see Resources on page ?).

Here's the definition of Unicode as provided by the Unicode home page: "Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language."

Unicode is already an essential part of the Java platform and other standards, such as XML, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, and so on.

What does Unicode do for me?

Computer programmers have long faced a number of problems that Unicode now provides a solution for:

- In old encoding schemes, the same value could represent different characters -- in Unicode, each value is unique.
- With Unicode, you do not need to label each string with the character set in which it is encoded.
- Unicode provides consistent:
  - Searching
  - Ordering and collation
  - Protection against data corruption when working with data from multiple languages

   - Rules for detecting characters that occupy multiple bytes

Unicode is no cure-all, though. There are a number of problems that Unicode does not solve:

- Text must still be input in the native code set and displayed and rendered to the screen with the proper fonts.
- You must still understand the different languages and their requirements.
- You must write your own bidirectional algorithms for displaying characters for right-to-left languages such as Hebrew or Arabic; the database does not take care of this by default.

Types of Unicode

There are two main flavours of Unicode being used today, UTF-8 and UCS-2 (aka UTF-16). Each has its own advantages and disadvantages.

UTF-8 stores all characters in a format from one to four bytes. All of the ASCII characters are stored as a single byte, and languages such as Japanese or Korean take three bytes per character. A few very, very rare characters take four bytes.

There are a number of advantages to using UTF-8 encoding in your database:

- The size of databases designed to store ASCII data will not need to change.
- The translation between ASCII and UTF-8 is very quick since they have similar encodings. ASCII is actually a subset of UTF-8: The binary representation of an ASCII character is equivalent to the binary representation of the same UTF-8 character, except that the UTF-8 representation will be preceded by 8 bits padded with zeros.
- UTF-8 data is stored as `CHAR` data in DB2, so no application changes are needed for `CHAR`-based apps.

There are disadvantages to UTF-8 as well:

- String manipulation and parsing becomes difficult because the string functions are byte based. This means that, for instance, `SUBSTR(stringName,1,2)` could return a partial character.
- The Java platform transmits data as UCS-2, so translation will always be needed when inserting or retrieving data from Java code.

UCS-2 (also known as UTF-16) stores all characters in a format on two bytes. There are a few very obscure characters that are stored as four bytes, but software vendors tend to ignore these rare exceptions and program based on a constant two-byte length.

There are a number of advantages to using UCS-2:

- The characters are all the same length, so manipulating and working with strings is simple.
- UCS-2 is the native encoding for the Java language, so no character encoding translation is required.

There are disadvantages as well:

- The size of ASCII data doubles, since all characters are two bytes. This will double the amount of disk space that you need.
- The characters are stored as `GRAPHICs` in DB2, so your stored procedures, functions, and applications will have to be redefined to accommodate the new type.

When to use which encoding?

Both UTF-8 and UCS-2 are used by the major software companies. It is up to you to decide which you should use each in your applications.

Using UTF-8

UTF-8 is used when space is a large concern. Converting a data warehouse from a non-Unicode standard to a Unicode database is a large task, since all the character data will have to be converted. Data warehouse systems also normally require a large amount of space. Translating a 2 TB database from ASCII to UTF-8 will keep it the same size. Using UCS-2, however, would double the size of the database and require an extra 2 TB of disk space! This would be an extremely expensive choice. It would be smarter to use UTF-8 to reduce your infrastructure costs.

Using UCS-2

UCS-2 makes programming easier, since all the characters are a fixed length. Your application can then easily translate your string manipulation functions to handle the two-byte characters. The database string manipulation functions for UCS-2 automatically treat the two bytes as a single character, so you can treat it just as you would an ASCII string. You would therefore not need to change any application logic.

UCS-2 is also the default standard for the Java language and for Microsoft Windows. This means that no encoding transformation on those platforms will be required if your data is stored as UCS-2.

How does DB2 use Unicode?

DB2 can store both Unicode encoding schemes in one database. All UTF-8 characters are stored as `CHAR`, `VARCHAR`, and `LONG VARCHAR` data types. UCS-2 characters are stored as `GRAPHIC`, `VARGRAPHIC`, and `LONG VARGRAPHIC` data types. Large Object Binaries, or LOBs, are also stored differently. UTF-8 LOBs are stored as Character Large Object Binaries, or CLOBs. UCS-2 LOBs are stored as Double-Byte Character Large Object Binaries or DBCLOBs.

Creating a Unicode database

It is simple to create a Unicode database. When you issue the DB2 `CREATE DATABASE` command, you have to specify the encoding scheme for the text in the database. If you do not specify the encoding scheme, then by default DB2 will use the OS's encoding scheme. If you wanted to store data in a Japanese encoding scheme, then you would provide DB2 with this information. Unicode is viewed as just another encoding scheme, albeit one that holds information for all languages.

Here's the command syntax:

```
--CREATE-DATABASE---database-Name---USING-CODESET--UTF-8---TERRITORY--territoryName--|
```

The *territory* that you use can be any valid territory code, such as `US` for the United States. Changing the territory value will not change the Unicode data. However, this piece of information is needed to keep the format of the `CREATE DATABASE` command consistent for Unicode databases as well. The `UTF-8` codeset variable must be written in all uppercase characters.

The following sample command line will create a Unicode database:

```
CREATE DATABASE uniTest USING CODESET UTF-8 TERRITORY us
```

How do I know if I have a Unicode database?

The encoding type of the database is recorded in the database configuration information. If you execute either of the following commands, you will be able to see what your database's encoding scheme is.

```
GET DATABASE CONFIGURATION FOR databaseName
GET DB CFG FOR databaseName
```

The database encoding information is at the top of the output and will look like this:

```
Database territory                  = US
Database code page                  = 1208
Database code set                   = UTF-8
Database country code               = 1
```

The codepage for UTF-8 is 1208 while the codepage of UCS-2 is 1200. The configuration will show UTF-8 as the codeset even if you only use UCS-2 data.

Converting data between Unicode and non-Unicode systems

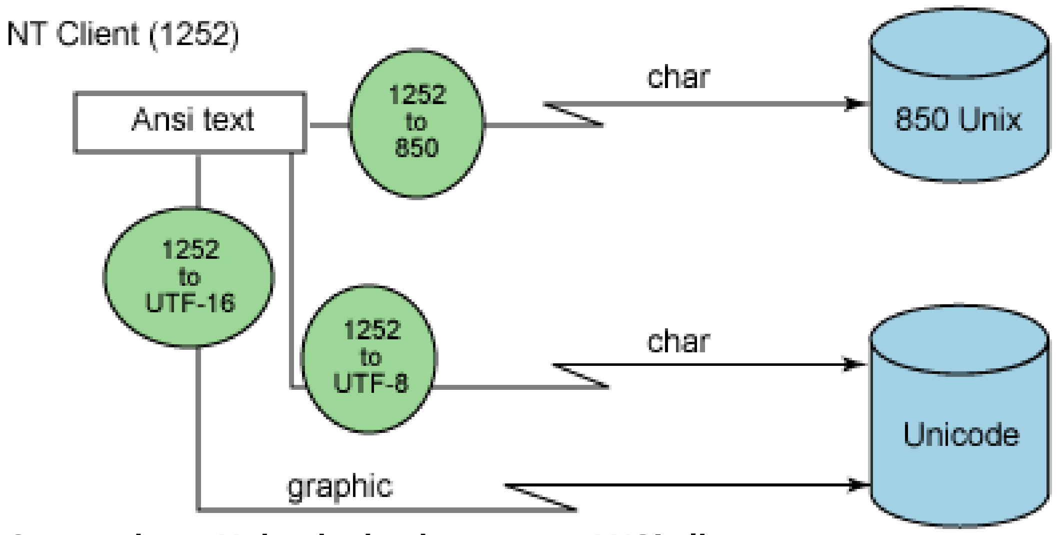
Often, your database may store information as Unicode, but your client system may still just be using a language-specific character encoding. This can result in data loss when data is transmitted back to the client: Characters in the database may not be representable in the client, or the character encoding values may mean something different on the different machines. The best practice is to write your client application to use Unicode as well. If that is not possible, then it is important to understand how data may be lost.

Specifying the client codepage

By default, the codepage of the client is equal to the codepage of the operating system that it is running on. This can be overridden by setting the DB2 CLI setting `DB2CODEPAGE` to be the codepage you want to use. This is useful if your application stores information that is in a different language, or if you are using a Unicode client.

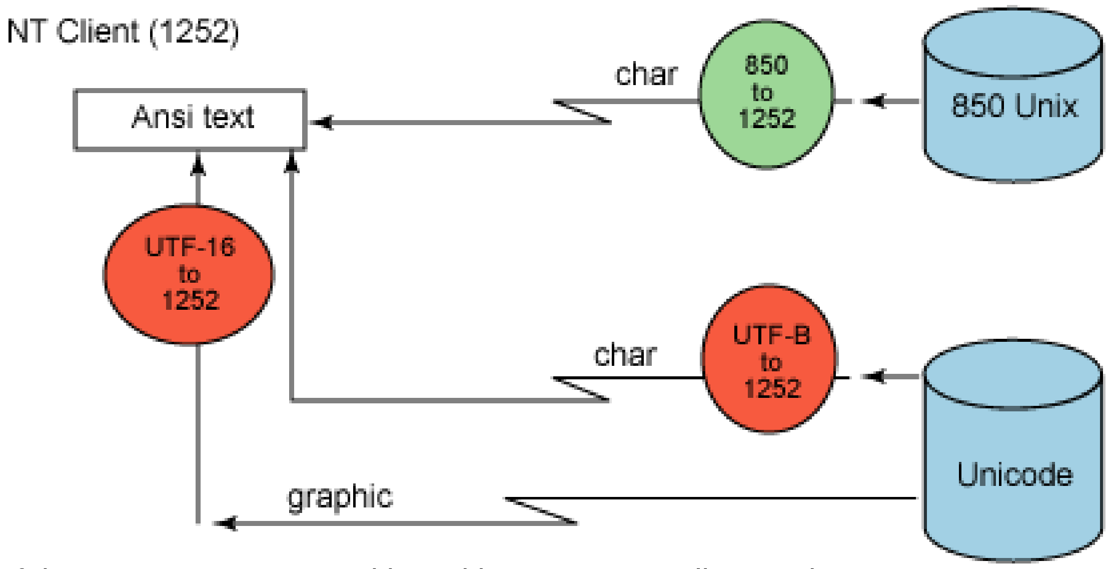
Connecting an ANSI client to a Unicode database

No data will be lost transmitting data from an ANSI client to a Unicode database. The codepage of the data will be translated from the ANSI codepage to the Unicode codepage when the data arrives at the server.

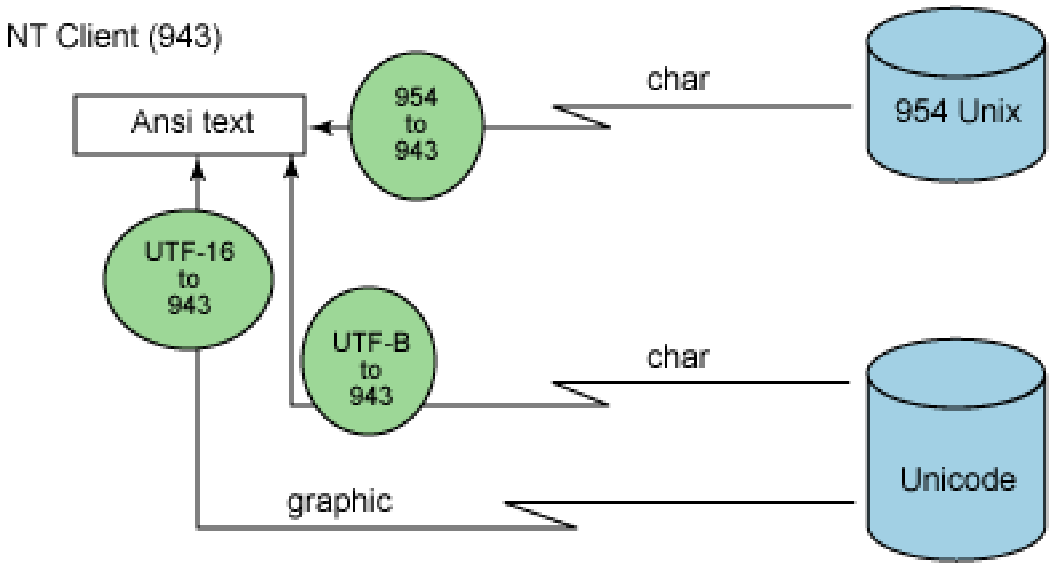


Connecting a Unicode database to an ANSI client

Unicode-encoded data may be lost when you transmit it to a non-Unicode client. If you have a client that is using the 1252 codeset, which is the default English codeset for Windows, you can only receive characters that are part of the ASCII codeset. If you sent Japanese data back to the client, it would come through as gibberish and would not be processed.



If, however, you were working with a Japanese client and you sent Japanese Unicode data back to it, then the data would be processed without a problem. The encoding would be translated at the server.



Collation order

The *collation sequence* for a language is the order in which letters will be sorted. This is commonly known as the order of the alphabet. The order of the characters varies from language to language, and is quite important when sorting character strings. Each character encoding has its own unique collation sequence, which is specific to the language. Some languages have multiple character encodings and each encoding may have a slightly different sort order. Consider, for example, an English encoding in which lowercase letters such as q are be sorted before uppercase letters such as Q. Another encoding may have Q first and q later.

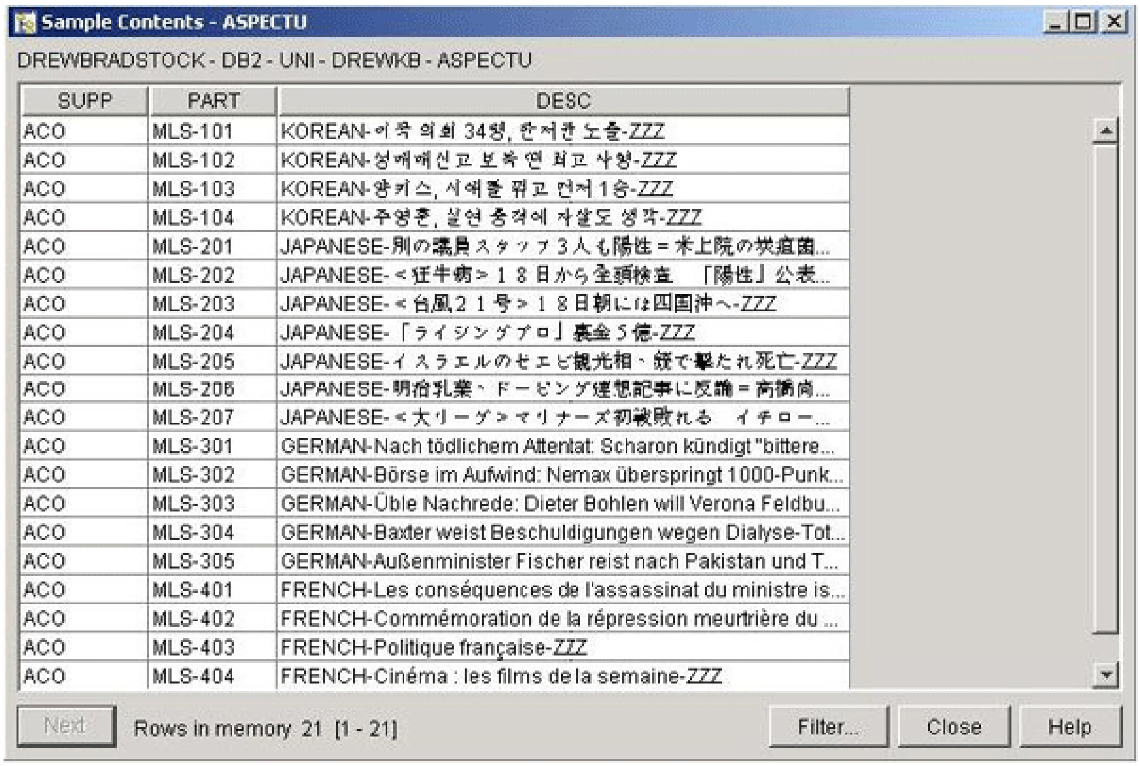
Unicode uses a single collation sequence for all languages. A number of prominent linguists worked on the collation sequence to generate one that is closest to the dominant sequence for each of the character sets stored in the encoding. You could consider this sequence as having multiple alphabets all in one set. The Korean character set would be sorted in one order, and Japanese in another. However, if you displayed strings with both languages, you would always receive the same sort order, since the characters are also part of a complete collation sequence for all of Unicode. The collation sequences for UTF-8 and UCS-2 are identical.

As a result, a Unicode DB2 database only supports a single collation sequence. DB2 allows you to manually override the collation sequence of an encoding, but doing so requires manually altering the file holding the collation sequence in a binary format. Doing this for the Unicode sequence would be far too complex and prone to errors, and would cause the database to behave differently from what people would expect and rely on in their applications.

Working with Unicode data

DB2 allows you to view the major language sets within the DB2 utilities. DB2 has the major fonts built in for the Unicode translations of the major languages. This allows you to view the values for Japanese, Chinese, Korean, and other languages even if you don't have a full font viewer installed on your system. These can only be displayed within DB2 if the font is not installed for your entire client.

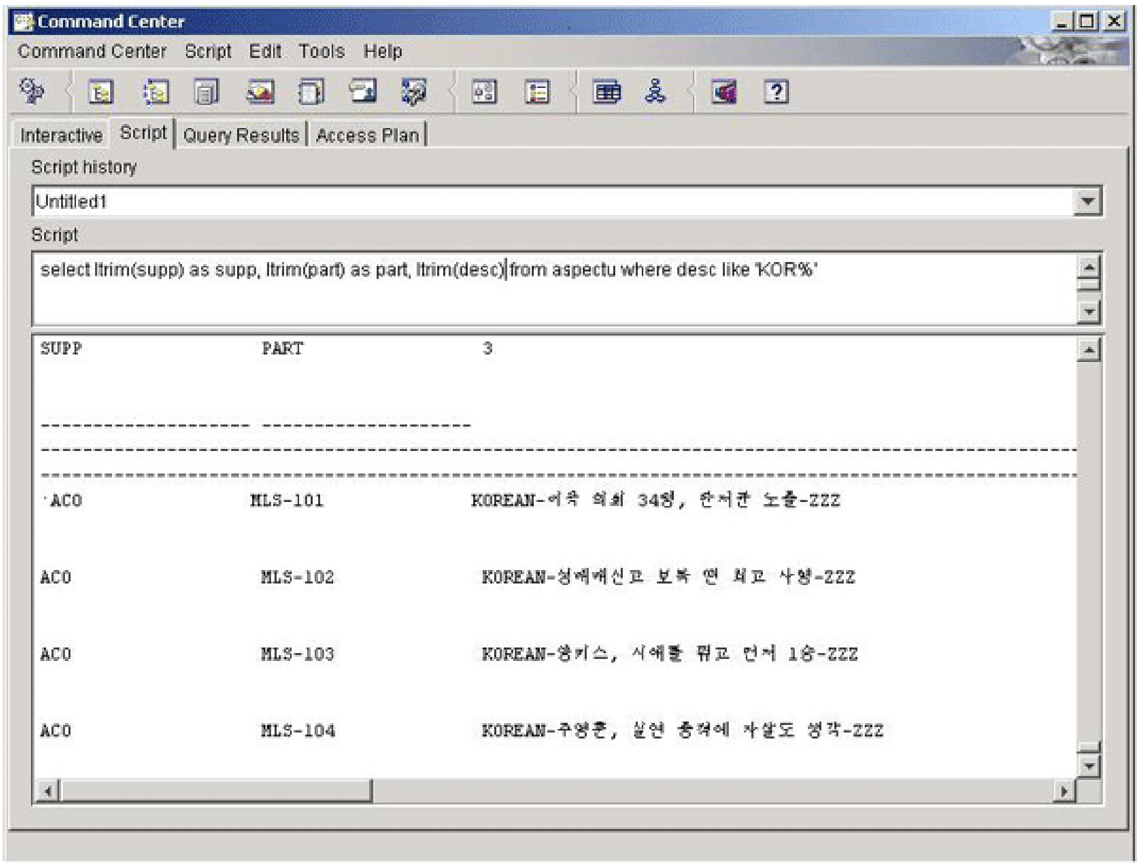
Here's what Unicode characters look like when displayed in the DB2 Control Center:


Sample Contents - ASPECTU
DREWBRADSTOCK - DB2 - UNI - DREWKB - ASPECTU

SUPP	PART	DESC
ACO	MLS-101	KOREAN-미국 의회 34명, 탄저균 노출-ZZZ
ACO	MLS-102	KOREAN-성매매신고 보복엔 최고 사형-ZZZ
ACO	MLS-103	KOREAN-양키스, 시애틀 꺾고 먼저 1승-ZZZ
ACO	MLS-104	KOREAN-주영훈, 실연 충격에 자살도 생각-ZZZ
ACO	MLS-201	JAPANESE-別の議員スタッフ３人も陽性＝米上院の炭疽菌...
ACO	MLS-202	JAPANESE-＜狂牛病＞１８日から全頭検査　「陽性」公表...
ACO	MLS-203	JAPANESE-＜台風２１号＞１８日朝には四国沖へ-ZZZ
ACO	MLS-204	JAPANESE-「ライジングプロ」裏金５億-ZZZ
ACO	MLS-205	JAPANESE-イスラエルのゼエビ観光相、銃で撃たれ死亡-ZZZ
ACO	MLS-206	JAPANESE-明治乳業、ドーピング捏造記事に反論＝両備尚...
ACO	MLS-207	JAPANESE-＜大リーグ＞マリナーズ初戦敗れる　イチロー...
ACO	MLS-301	GERMAN-Nach tödlichem Attentat: Scharon kündigt "bittere...
ACO	MLS-302	GERMAN-Börse im Aufwind: Nemax überspringt 1000-Punk...
ACO	MLS-303	GERMAN-Üble Nachrede: Dieter Bohlen will Verona Feldbu...
ACO	MLS-304	GERMAN-Baxter weist Beschuldigungen wegen Dialyse-Tot...
ACO	MLS-305	GERMAN-Außenminister Fischer reist nach Pakistan und T...
ACO	MLS-401	FRENCH-Les conséquences de l'assassinat du ministre is...
ACO	MLS-402	FRENCH-Commémoration de la répression meurtrière du ...
ACO	MLS-403	FRENCH-Politique française-ZZZ
ACO	MLS-404	FRENCH-Cinéma : les films de la semaine-ZZZ

Rows in memory 21 [1 - 21]

The data can also be displayed in the Command Window, as the figure below illustrates. This allows you to run your queries and ensure that they are returning the correct values, even if the results are in another language. This can be quite helpful when you are troubleshooting applications that display multiple languages at the same time.


Command Center
Command Center Script Edit Tools Help
Interactive Script Query Results Access Plan
Script history
Untitled1
Script

```
select ltrim(supp) as supp, ltrim(part) as part, ltrim(desc) from aspectu where desc like 'KOR%'
```

```
SUPP                 PART                 3


-------------------- --------------------
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
·ACO                MLS-101              KOREAN-이혹 의회 34명, 한지관 노출-ZZZ


ACO                  MLS-102              KOREAN-성매매신고 보복 연 처고 사형-ZZZ


ACO                  MLS-103              KOREAN-왕키스, 시애틀 뛰고 먼저 1승-ZZZ


ACO                  MLS-104              KOREAN-주영훈, 살인 충격에 자살도 생각-ZZZ
```

Further reference

You can search the DB2 Information Center (see Resources on page 52) for the following keywords for more information about Unicode:

- Unicode database
- UTF-8
- UCS-2
- Codepage
- Territory

Section 9. Using DB2 performance features

Using DB2 performance features

If you want your application to perform at its best on DB2, you should become familiar with the performance features that have been built into it for application development. As an application developer, you probably don't have the time to learn the ins and outs of the entire database, but by quickly reading through the release notes of the latest database version, you may find some new functions that can really improve your application with little effort.

In this final section of our tutorial, we'll look at two such features: *buffered inserts* and `MERGE` *statements.*

Buffered inserts

Often, an application requires a large amount of information to be loaded into the database. You may be fortunate to have the data in a text file, in which case you could use the DB2 `LOAD` command to quickly load the data into tables. You may, however, want to still be able to alter the table as you load the data, which `LOAD` does not allow you to do. In that case, you could use the `IMPORT` command instead.

Often, though, the data being loaded in is based on user selections, so it is not in a flat file but instead in your application's memory. Using a series of individual `INSERT` statements in this situation can be quite costly, because each will be sent to the database and executed, and the result code will be sent back, before the next insert can occur. In applications where there is a long distance between the client and database, the extra network latency can cause a serious performance problem. It would be much more efficient if you could join together a series of `INSERT` statements into one group. This can be achieved by using *buffered inserts.*

A buffered insert occurs when you send a group of `INSERT` statements over to the database server from a client as one large group. To take advantage buffered inserts, the application must be prepared or bound using the `INSERT BUF` option. Buffered inserts are normally used when a single `INSERT` statement is repeated multiple times in a loop, and no other database modification statements are issued in between the inserts. Normally, the `INSERT` statements reference a value that changes as the loop is iterated. Instead of sending the `INSERT` statements as individual statements, they are all grouped together in a table queue and run as one unit.

Error handling

It is important to understand that, because the `INSERT` statements in a buffered insert are all sent and executed as a single group, you will be unable to tell which of the inserts failed if an error occurs. The entire batch of `INSERT` statements will fail if even a single statement fails. The error is also not reported immediately, but instead reported when the first statement that is an `UPDATE` or `DELETE`, or an `INSERT` on a different table, is executed.

Viewing of buffered rows

An inserted row may not be immediately visible to `SELECT` statements issued after the `INSERT` by the same application program, if the `SELECT` is executed using a cursor.

Buffered inserts: An example

The following example is taken from the DB2 Information Center documentation on buffered inserts and illustrates how they can be used.

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
  READ VALUE OF hv1 FROM A FILE;
  EXEC SQL INSERT INTO t2 VALUES (:hv1);
  IF 1000 INSERTS DONE, THEN DO
     EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
     RESET COUNTER;
  END;
END;
EXEC SQL COMMIT;
```

Now, suppose our file contains 8,000 values, but value 3,258 is not legal (it may, for example, include a unique key violation). Each set of 1,000 inserts results in the execution of another SQL statement, which then closes the `INSERT INTO t2` statement. During the fourth group of 1,000 inserts, the error for value 3,258 will be detected. It may be detected after the insertion of more values (not necessarily the next one). In this situation, an error code is returned for the `INSERT INTO t2` statement.

The error may also be detected when an insertion is attempted on table t3, which closes the `INSERT INTO t2` statement. In this situation, the error code is returned for the `INSERT INTO t3` statement, even though the error applies to table t2.

The MERGE command

A common task in an application is the insertion of data into a table. Here's a common scenario: An application wants to update or insert a row into a table depending on whether it is already there. If that row already exists in the table, then the row is updated. If not, then the a new row is inserted. This may sound like a simple operation, but it can often involve many SQL statements and complicated logic. A new SQL command, `MERGE`, was added in DB2 8.1. This command allows you to perform both commands in one statement. This greatly reduces the complexity of your application and passes the logic into the database engine.

The example below shows how the `MERGE` command could be used to control the insert or update of a group of rows into the account table from the transaction table.

```
MERGE INTO account AS a
USING (SELECT id, sum(balance) sum_balance FROM transaction
       GROUP BY id) AS t
ON a.id = t.id
WHEN MATCHED THEN
   UPDATE SET
      balance = a.balance + t.sum_balance
WHEN NOT MATCHED THEN
   INSERT (id, balance) =
          (t.id, t.sum_balance);
```

The complete syntax of the `MERGE` statement is quite flexible, and therefore can be quite complex if you choose. The command was added in DB2 V8.1 Fixpak 2 and the syntax diagram is in the documentation for DB2 V8.1 Fixpak 2 or later.

Further reference

You can search the DB2 Information Center (see Resources on page 52) for the following keywords for more information about these performance-enhancing features.

- Buffered insert
- `MERGE`

Section 10. Conclusion

Summary

This tutorial was written to help you with the sixth part of the DB2 Application Development exam (Exam 703). By reviewing this information, you should now have a greater understanding of a number of key concepts related to running your application on DB2. This tutorial was designed as an overview to these concepts. If you are looking for more in-depth knowledge, then the DB2 Information Center (see Resources on page 52) is the ideal place to look.

To make writing your application easier, it is important that you understand how the database is going to interact with your program, and how you can utilize the functionality it has built into it. By leveraging your new knowledge of database concurrency and advanced techniques, you should be well on your way to getting great performance of your application on DB2.

Resources

This tutorial will help you study for the application development exam, but there is only so much space and time to explain the many concepts. Each of the sections has a panel at the end that refers to you to keywords that you can look up in the DB2 documentation for further information.

The best source of information for DB2 is the new Information Center. This Information Center was added in DB2 V8.1 Fixpak 4 and has an entirely new interface written using the Eclipse programming framework. It is quick and easy to use, and the search utility built into it is far better than any of the previous incarnations. It also has links to other sources of DB2 information on the Web, such as Google and the DB2 Technotes.

All of my references at the end of the sections refer to this source and not to the DB2 documentation included with the product itself. The new Information Center is written by the DB2 Information Development team and is the new official reference for DB2 help. Go out and try it!

The *DB2 Information Center* is also available on the Web.

If you need to know more about Unicode, check out the *Unicode home page* (http://www.unicode.org/) .

- For more information on the DB2 UDB V8.1 Family Application Development Certification (Exam 703), see *IBM DB2 Information Management -- Training and certification* (http://www.ibm.com/software/data/education/) for information on classes, certifications available and additional resources.
- As mentioned earlier, this tutorial is just one tutorial in a series of seven to

help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:

1. *Database objects and Programming Methods*
2. *Data Manipulation*
3. *Embedded SQL Programming*
4. *ODBC/CLI Programming*
5. *Java Programming*
6. Advanced Programming
7. User-Defined Routines

- Before you take the certification exam (DB2 UDB V8.1 Application Development exam, Exam 703) for which this tutorial was created to help you prepare, you should have already taken and passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:
    - DB2 planning
    - DB2 security
    - Accessing DB2 UDB data
    - Working with DB2 UDB data
    - Working with DB2 UDB objects
    - Data concurrency
- Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:
    - Server management
    - Data placement
    - Database access
    - Monitoring DB2 activity
    - DB2 utilities
    - Backup and recovery

Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial

generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .